Technical Report 1672
September 1994

Scalable Programming Environment

Perry Partow
Dennis Cottel

Sponsored by Office of Naval Research

EXECUTIVE SUMMARY

OBJECTIVE

This report describes the Scalable Programming Environment (SPE), which provides programmers with a transparent way of creating scalable parallel applications for large-grained parallel computer architectures.

APPROACH

The SPE was developed on the Intel Paragon to support the Hybrid Digital Optical Processor (HyDOP), a real-time acoustic signal processing application for undersea surveillance, sponsored by the Office of Naval Research (ONR-321). All the scalable and reconfigurable needs of HyDOP have been incorporated in a library of general programming calls. The development and testing of the SPE evolved as the HyDOP project dictated. The Intel Paragon was made available by the DoD HPC Modernization Program.

The SPE was designed with generality in mind, so that in addition to meeting the needs of HyDOP, it could be used in other similar types of applications. At least one other project has already begun to use the SPE and is beginning to influence the SPE development.

RESULTS

The SPE, which has been designed primarily to support data-flow processing applications, allows programs to be scaled to execute on any number of processing nodes while requiring no changes to the compiled binary code. The user is provided with a set of high-level message-passing routines which can be used to connect multiinstanced heterogeneous programs in a system. The SPE library routines hide the intricacies of how the parallel programs communicate. The details of the connections are specified in text files. The SPE allows individual programs to be coded without knowledge of other parts of the system and thus allows systems to be quickly built, modified, or scaled without program recompilation.

At the time this report is being written, the SPE is still under development. All the significant parts have been implemented and tested on an Intel Paragon XP/S 25. Although the current implementation interfaces to the operating system using Intel-specific NX calls, it should be portable to the emerging Message Passing Interface standard or to other vendor-specific parallel operating system interfaces based on message passing.

The SPE has been successfully used by the HyDOP project, and all newly developed HyDOP programs are currently using the SPE. Use of the SPE has provided more rapid program development and system integration. The current HyDOP subsystem is scalable and reconfigurable.

The SPE is also being evaluated for use in synthetic aperture radar (SAR) image-formation processing for an ARPA-sponsored project. The goals of this project include demonstrating the ability to perform SAR processing in real time on the Paragon in a scalable implementation which has potential for portability to other parallel processors. It is expected that by using the SPE, the SAR development time will be significantly reduced.

CONTENTS

FIGURES

1.0 INTRODUCTION

The Scalable Programming Environment (SPE) is a programming environment and system interface which was developed by the Hybrid Digital Optical Processor (HyDOP) project, sponsored by the Office of Naval Research (ONR-321), to help build large scalable real-time systems in a research and testbed environment. It provides the user with the ability to build and modify scalable systems quickly using both function- and data-domain decomposition methods.

The SPE is a data-flow parallelizer. It loads and runs heterogeneous programs on multiple sets of nodes and provides the scalable data-path connections needed for unrelated parallel programs to communicate.

Each program in a *system*, or an *application*, executes on a set of nodes and performs a different function. Each node for a given program executes the same code, called an *instance* of the program. Each instance of a program is expected to work on a different piece of the data for the given program. The SPE provides the complex high-level message-passing routines which are needed to interconnect different programs and instances of programs into a system.

Currently the parallel computing industry does not provide a standard set of high-level message-passing routines to systematically interconnect multiinstanced heterogeneous programs in a system. These systems must be built with the details of the message passing visible to the application programmer. A multi-instanced program is developed having to know the intricacies of the other programs it is connected to, how the data are shaped on the other end of the communications path, how it will control the flow of the data it receives, how it will buffer and transform the data once received, and how it will present the data synchronously to multiple instances of itself.

Figure 1a shows the level of message-passing detail that a traditionally developed multiinstanced program sees in a heterogeneous system. Each instance of program B must be aware of where it gets its data from. Each instance must be aware of how the data is shaped at the other end. Each instance will have to control the flow of data it receives (request messages). If program B receives messages from two or more programs (not shown), then each instance will have to guarantee that it receives all messages in the same order as other instances (controlled by the synchronizing messages).

This level of message-passing detail is beyond the level that an application programmer should have to worry about. Furthermore, it is time-consuming and prohibits prototyping large systems or modifying existing ones. It has kept system designs from emerging beyond the multi-instanced single-program standard predominantly used today.

The SPE has been developed to hide this level of message-passing detail. Figure 1b shows the view that an SPE program sees when communicating data. Programs communicate with each other through ports

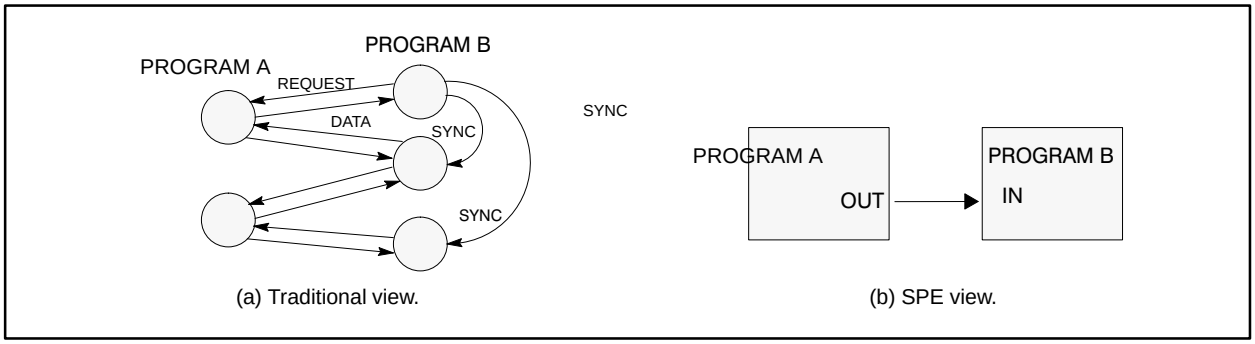


Figure 1. Message-passing between heterogeneous programs.

connected by nets. Each is unaware of which program it talks to, the number of program instances at the other end, the number of program instances at its own end, how the data are buffered, how the flow of data is controlled, and how the data are synchronized.

Furthermore, because this level of detail is hidden from the program, new systems can be quickly built and old ones quickly modified. Programs of a system can be scaled to run on any number of processing nodes while requiring no changes to source code. Programs can be disconnected and reconnected in different ways to modify the function of a system.

The SPE message-passing routines have also been built to utilize resources efficiently. They have been designed to overlap processing with communication, minimize buffer space, avoid extra copying of data, and minimize the number of messages.

The SPE also includes other features useful in a parallel programming environment, such as dynamic run-time control of diagnostic flags, execution parameters, performance monitoring, logging, and error reporting.

The SPE provides:

1 A loader, which allows the user to define, load and run parallel programs on scalable sets of nodes without the need to recompile. New systems can be built or modified by changing a *System Definition* file. Systems can be easily run on varying numbers of nodes to change system performance or meet constraints of the hardware system.

2 High-level message-passing routines to transfer data between programs running on differing numbers of nodes. Multiple programs are interconnected with data-flow-type connections, which hide the parallelism of the system within the connections. The message-passing routines provide the scatter–gather-type operations needed to pass data between programs running on differing numbers of nodes, provide internal synchronization controls to make messages received by programs synchronous to every instance of a program, and provide first in, first out (FIFO) data buffers so that programs sending and receiving messages between each other can work on different-size data blocks.

3 A debugging environment for performance monitoring, logging, and error reporting. Debugging a parallel application requires a user interface which deals with multiple programs and multiple instances of programs. The SPE provides *report()*, a *printf()*-like call, which conditionally writes to standard output based on a run-time parameter which can be unique to each *report()* call. Other routines allow the user to view data as one image across all instances of a program.

4. A global database for the storage of symbolic names with their associated values. Parameter values within a system can be stored through the user interface interactively at run time so that application code need not be recompiled when values are changed or new ones added.

At the time this report is being written, the SPE is still under development. All the significant parts have been implemented and tested on an Intel Paragon XP/25. Although the current implementation interfaces to the operating system using Intel-specific NX calls, it should be portable to the emerging Message Passing Interface standard or to other vendor-specific parallel operating system interfaces based on message passing.

This report is organized as follows: Section 2 describes the user interface. It also describes the input text files through which the user defines an SPE application. These files define each SPE program, describe

how they are connected to form a system, and initialize database variables that can be used by the programs. Examples are shown for each of the files, and rules are provided describing the grammar.

Section 3 describes how parallel programs communicate through ports. Different types of ports are described which define how data are scattered and gathered when communicated between parallel programs. Also discussed is how data are buffered and synchronized between programs.

Section 4 describes the programming interface. Different SPE routines are described and examples are provided showing how they are used in typical application.

Section 5, which will be expanded at a later date, shows how to compile and run an SPE application.

Appendix A shows the decomposition algorithm used by the SPE when gathering or scattering data over a port. Appendix B shows predefined database variables which can be used by the user to obtain diagnostic information from the SPE. Appendix C shows the key words recognized by the SPE when interpreting the input text files. Appendix D defines each of the SPE programming calls.

2.0 USER INTERFACE

The user interfaces to the SPE through a *System Definition* file, *Program Definition* files, and *Database Startup* files. The System Definition file and Program Definition files describe in two levels how each program in a system interfaces to other programs in the system, and how each program interfaces to its outside world. The Database Startup files are used by the SPE to enter variables into a global database which can be used by each program in the system.

A user loads and runs an SPE application by executing `spe`. As shown in this command line description, the `spe` must be supplied with a path name to the System Definition file to run an application:

```
% spe –pn partition –on 0 –s sys_def_filename
      [[-d database_startup_filename]...] [-l log_filename]
      [-ident] [–identall] [–noload] [-portmap]
      [[–Dname]...] [[–Dname=def]...]
```

When `spe` starts, it reads the System Definition file and Program Definition files. From these files, it determines the configuration of the system, and loads the programs specified in the System Definition file onto nodes of the target hardware. `spe` then creates and downloads a unique *port map* to each program instance in the system, which describes exactly how each program is connected to the other programs of the system. `spe` then reads the Database startup files as specified on the command line and initializes the SPE database. Once all the programs have received their unique port map and the SPE database is initialized, the programs are ready to run. `spe` interacts with the user through standard input and output.

The `spe` preprocesses all input files through a C preprocessor (GNU `cpp`). This allows for file inclusion and macro expansion as specified by the C language. The *–Dname* and *–Dname=def* arguments to `spe` are passed directly onto the preprocessor and behave as described in the manual pages for `cpp`. The format of the input files, after preprocessing, must conform to the file formats described in the following sections.

2.1 SYSTEM DEFINITION FILE

The System Definition file defines which programs are used in a system, how many nodes each program will run on, and how programs are interconnected. `spe` uses it to determine what programs will be loaded on what nodes and to make a unique port map for each program instance. Figure 2 is an example of a System Definition file.

The System Definition file specifies the path name to each program which is to be loaded by `spe`, the number of nodes that each program will run on, and a path name to the Program Definition file for each program which will be loaded. After reading the System Definition file, `spe` then reads each Program Definition file so that it can completely determine the port map for each program in the system. The port map describes for each program instance what portion of the problem it will work on and how it is connected to other programs in the system. `spe` determines which portion of the data each program instance will work on based on the Program Definition file for a program and based on the number of nodes the program is specified to run on.

```
// File: system/receiver
//
// Key Word  #Nodes Program        Prog_Def_Path         Executable_Path
// ------------------------------------------------------------------------
   PROGRAM   1    ship            "def/ship"            "bin/ship"
   PROGRAM   1    hydrophone      "def/hydrophone"      "bin/hydrophone"
   PROGRAM   1    display         "def/display"         "bin/display"
   PROGRAM   10   beamformer      "def/beamformer"      "bin/beamformer"
   PROGRAM   15   matched_filter  "def/matched_filter"  "bin/matched_filter"

// Key Word  Input_Port             Buffer
// ------------------------------------------------------------------------
   BUFFER    matched_filter:gain    1
   TRANSPOSE beamformer:d_in
   FUNNEL    display:elem

// Key Word  Net_list
// ------------------------------------------------------------------------
   NET       ship:gain, matched_filter:gain, beamformer:gain
   NET       ship:ctl, beamformer:ctl
   NET       hydrophone:elem, beamformer:d_in, display:elem
   NET       beamformer:d_out, matched_filter:d_in
```

Figure 2. Example of a System Definition file corresponding to the system shown in figure 3.

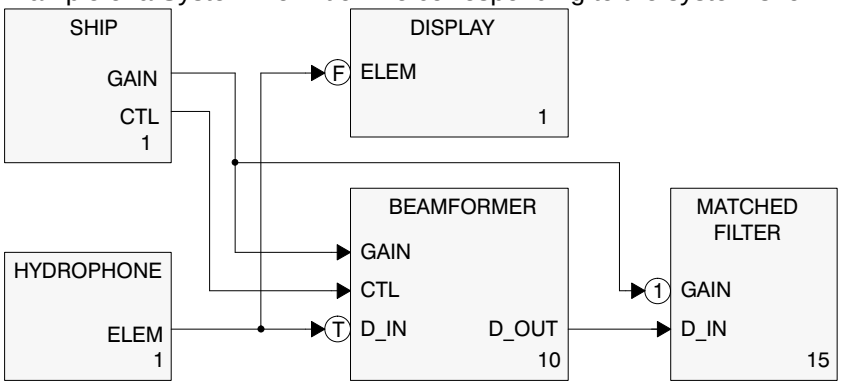


Figure 3. Example showing an implementation of an acoustic receiver system.

The System Definition file also specifies the *net lists* which connect *ports* of each program in the system. Programs communicate by passing input and output data through their ports. Data written to an output port of a program are sent to the input ports on the net list to which the output port is connected. A program is unaware of where its data go or where the data come from. This basic principle of the SPE allows users to build modular systems which can be quickly modified or extended to meet the changing needs of an application, and also promotes the reuse of software when new applications are built.

The System Definition file also specifies whether the data received by an input port need to be *transposed, funnelled*, or *additionally buffered* to meet the input requirements of a program. Data must be transposed when two connected programs need to process their data along different decomposition lines (such as in a two-dimensional fast fourier transform (FFT)). Data must be funnelled when an input port consumes less data than is provided by the output port to which it is connected. Additional buffering on an input port may be needed to improve efficiency by increasing the amount of data the port can store.

The System Definition file of figure 2 is graphically represented in figure 3. Each box represents a different program in the system, with a number in the lower right-hand corner indicating the number of nodes it runs on. Each box has named ports through which it communicates data to other ports on its net. Input ports which are additionally buffered, transposed, or funnelled have an attached bubble containing an integer, the letter "T" or the letter "F," respectively.

The rules for making a System Definition File are as follows:

1. Tabs and blanks are white spaces and are used to delimit lines into tokens interpreted by the parser. Tokens are either key words, identifiers, integers, or strings. Backslash (\) can be used to continue the end of a line. The rest of the line after a double forward slash (//) is ignored as a comment.
2. Tokens are either key words, identifiers, integers, or strings. Key words which can be used in the System Definition file include `PROGRAM`, `BUFFER`, `FUNNEL`, `TRANSPOSE`, `NET`, and `NA`, and may be written in uppercase or lowercase letters. Identifiers must be an allowable C language identifier, up to 31 characters, and cannot include any of the key words found in Appendix C. Strings are enclosed in double quotes.
3. Each line must begin (ignoring white spaces) with `PROGRAM`, `BUFFER`, `FUNNEL`, `TRANSPOSE`, or `NET`. These key words determine the format of the rest of the line.
4. Each line starting with `PROGRAM` specifies a program used in the system. The first field is an integer specifying the number of nodes the program runs on. The second field is an identifier specifying the symbolic name by which the program will be referenced in other parts of this file and the Database Startup files. The third field is a string that specifies the path name to the Program Definition file. The fourth field is a string that specifies the path name to the executable which runs on the nodes. The path name can include program arguments (separated by spaces within the string).
5. Each line starting with `BUFFER` specifies an input port that needs extra buffering to store data received on that port. The amount of buffering needed is expressed in integer blocks of input data (defined by the input port).
6. Each line starting with `TRANSPOSE` specifies an input port that must have its data transposed when received.
7. Each line starting with `FUNNEL` specifies an input port that will consume less data than produced by the output port it connects to. A variable in the database will control what subset of the data is actually passed.
8. Each line starting with `NET` specifies a list of ports which are connected together in a net. Ports are formatted as *`program:port`*, where *`program`* and *`port`* are replaced by identifiers. The first port in a net must be an output port. The remaining ports must be input ports. Ports do not have to be connected to a net. (There is an SPE call that can check if a port is connected, *portisconnected()*.)
9. `Control` ports can only be connected to `Control` ports. (`Control` ports will be described later in Section 3.4.)
10. The *`rows`* dimension of all nontransposed and nonfunnelled input ports on a net must agree with the output port to which it connects. The *`rows`* dimension of all transposed input ports on a net must agree with the *`columns`* dimension of the output port to which it connects. The *`rows`* dimension of all funnelled input ports on a net must be less than or equal to the rows of dimension of the output port to which it connects. (The *`rows`* and *`columns`* dimensions of a port are described later.)

2.2 PROGRAM DEFINITION FILE

The Program Definition files define each program's input and output. They are used, along with the System Definition file, to make a unique port map for each program and instance in the system. Each Program

Definition file defines only the input and output for its own program. There is no information in it defining what the program is connected to. For each use of a program in a system, there can be a different Program Definition file.

A Program Definition file defines each port of a program. A port is defined by its *direction*, *configuration*, *type*, *array size*, *element size*, *stripe overlap*, and *block overlap*.

The port *configuration* labels the port's definition as a configuration of the port. A port can have multiple definitions, with a different configuration label for each. The configuration label is used by the program to do dynamic reconfiguration of the port at run time (see Section 4.1.6). Each configuration for a port must be specified on a separate line.

The port *direction* indicates whether a port is an input or output port. Other fields of a port's definition can or cannot be specified, depending on its direction.

The port *type* describes whether or how data will be decomposed among a program's instances when data are received or sent from a program, how the data will be buffered between programs, and how the flow of data will be controlled between programs. The port type can be defined as *striped*, *replicated*, or *control* (see Section 3.0 for a description of each).

The port *array size* defines the size of the data a port receives or sends. It is specified only for replicated and striped port types. If specified, it defines a port by two dimensions, rows and columns. It need not actually be two-dimensional, but to the SPE it must be described as such (i.e., `[1][1]`, `[1][5],` and `[5][1]` are valid).

The port array size defines the size of the data before decomposition. That is, if the port is striped, then each instance of a program will see only its portion of the data. If the port is replicated, then each instance will see all the data. For striped ports, the data are decomposed across rows of the array (see Section 3.1).

If a port is replicated, then the rows dimension can be any value. If it is striped, then the number of rows must be greater than or equal to the number of program instances.

The port *element size* defines the size of each element of the data when the port array size is specified. The port element size will vary depending on the data which are processed (i.e., complex, real, etc.).

The port *stripe overlap* defines how many rows of overlap to use when decomposing data across a striped input port. For striped ports, the SPE decomposes the data across rows of the array. When the data are overlapped adjacent program instances share common rows of the data between them. The port stripe overlap can be specified only for input striped ports. Appendix A specifies the algorithm used for decomposing overlapped and nonoverlapped data over program instances.

Shown in figure 4 are examples of Program Definition files that could have been used in the receiver system example in figure 3.

The rules for making a Program Definition file are as follows:

1. Tabs and blanks are white spaces and are used to delimit lines into tokens interpreted by the parser. Tokens are either key words, identifiers, integers, or strings. Backslash (\) can be used to continue the end of a line. The rest of the line after a double forward slash (//) is ignored as a comment.
2. Tokens are either key words, identifiers, integers, or strings. Key words which can be used in Program Definition files include `PORT`, `INPUT`, `OUTPUT`, `STRIPED`, `REPLICATED`, `CONTROL`, and `NA`, and may be written in uppercase or lowercase letters. Identifiers must be an allowable C language identifier, up to 31 characters, and cannot include any of the key words found in Appendix C. Strings are enclosed in double quotes.

```
// File: def/beamformer
//                 Config_  Direc              Array_       Elem_   Striped_   Block_
// Key Word Port   uration  tion    Type       Size         Size    Ovlp       Ovlp
// ------------------------------------------------------------------------------
   PORT    gain    standard INPUT   REPLICATED [1][1024]    4       NA         0
   PORT    ctl     standard INPUT   CONTROL    NA           NA      NA         NA
   PORT    d_in    standard INPUT   STRIPED    [100][1024]  8       0          0
   PORT    d_out   high_res OUTPUT  STRIPED    [100][512]   8       NA         NA
   PORT    d_out   low_res  OUTPUT  STRIPED    [50][512]    8       NA         NA

// File: def/hydrophone
//                 Config_  Direc              Array_       Elem_   Striped_   Block_
// Key Word Port   uration  tion    Type       Size         Size    Ovlp       Ovlp
// ------------------------------------------------------------------------------
   PORT    elem    standard OUTPUT  STRIPED    [256][100]   8       NA         NA

// File: def/ship
//                 Config_  Direc              Array_       Elem_   striped_   Block_
// Key Word Port   uration  tion    Type       Size         Size    Ovlp       Ovlp
// ------------------------------------------------------------------------------
   PORT    ctl     standard OUTPUT  CONTROL    NA           NA      NA         NA
   PORT    gain    standard OUTPUT  REPLICATED [1][1024]    4       NA         NA

// File: def/display
//                 Config_  Direc              Array_       Elem_   Striped_   Block_
// Key Word Port   uration  tion    Type       Size         Size    Ovlp       Ovlp
// ------------------------------------------------------------------------------
   PORT    elem    standard INPUT   STRIPED    [4][512]     8       0          0

// File: def/matched_filter
//                 Config_  Direc              Array_       Elem_   Striped_   Block_
// Key Word Port   uration  tion    Type       Size         Size    Ovlp       Ovlp
// ------------------------------------------------------------------------------
   PORT    d_in    high_res INPUT   STRIPED    [100][2048]  8       4          0
   PORT    d_in    low_res  INPUT   STRIPED    [50][2048]   8       4          0
   PORT    gain    standard INPUT   REPLICATED [1][1024]    4       NA         0
```

Figure 4. Possible Program Definition files for the receiver system of figure 2.

3. Each line starting with `PORT` specifies the definition of a port. The fields are *Port, Configuration, Direction, Type, Array Size, Elem Size, Striped Ovlp*, and *Block Ovlp*. When a field is not allowed to be specified, it must contain `NA`, for "not applicable."
4. The *Port* field is an identifier specifying the name by which the port will be referenced.
5. The *Configuration* field is an identifier which identifies the port definition as one configuration of the port. It must be unique for every definition of the port. The first definition of a port is the configuration which the SPE uses at load time. Alternate configurations for the same port must have the same values for port *Direction* and port *Type*. All other fields can be different.
6. The *Direction* field must be specified as `INPUT` or `OUTPUT`.
7. The *Type* field must be specified as `CONTROL`, `STRIPED`, or `REPLICATED`.
8. The *Array Size* and *Elem Size* fields must be and can only be specified for striped and replicated ports. The format for the *Array Size* field when specified is *`[rows][columns]`*, where *`rows`* and *`columns`* are integers. The *Elem Size* field is the number of bytes for each element.
9. The *Striped Ovlp* field must be and can only be specified for input ports that are striped. It must be an integer less than the number of rows of the input data.

10. The *Block Ovlp* field must be and can only be specified for input ports that are striped or replicated. It must be an integer less than the number of columns of the input data. Usage of the Block Ovlp field will be described later in Section 3.2.

2.3 DATABASE STARTUP FILE

The SPE provides a global database to store symbolic names with their associated values. Programs are able to use the database to store such things as signal processing parameters, function control and switches, display parameters and control flags, and report and logging flags. Typically these values are found in include files and are shared among programs. If instead they are stored in a global database, then when the values are changed or new ones added entire sets of programs need not be recompiled.

The other uses of the global database are to communicate values from the user interface to a program or from one program to another. From the user interface, symbolic names and their associated values can be assigned to different programs or to specific instances of programs. For example, one may want to set a debugging or logging flag for a specific instance of a particular program, or may want to set a program variable to different values for each use of the program (i.e., a program which can do multiple functions).

A program can also store or access data in the database from the program interface. However, unlike the user interface, a program cannot assign a variable to a specific program or instance of a program. When a program sets a variable in the database, it applies it to all programs which have *registered* the use of that variable.

Programs tell the database that manager they are interested in a variable by registering for it. When a variable in the database is modified via the program interface, that variable is updated automatically in all programs which have registered for it. This means that programs within a system can be developed without having to know the requirements of other programs. Details about the program interface will be discussed later in Section 4.2.

The user interface allows the user to provide to `spe` a list of Database Startup files, which contain an initial set of symbolic names and associated values for the system. The user can also, through the course of a run, provide new names and values, or modify existing ones.

The Database Startup files contain a list of variables and associated values used by different programs in the system. Because a system is a set of programs, and each program is a set of instances, a symbolic name can have a different value for each program and instance in a system. Each line in the Database Startup file allows a variable to be assigned to all instances of a specific program, to a specific instance of a program, or to all instances of all programs. The same variable can be specified more than once (on a different line). An example of a Database Startup file is given in figure 5.

```
// File: database/receiver
//
// Key Word Name                Type(Value)       Program(instance)
// ------------------------------------------------------------------------------
   VAR      number_of_widgets   100               display            //integer
   VAR      narrow_band         TRUE              matched_filter     //integer
   VAR      speed_of_sound      1500.0                               //floating-point
   VAR      input_filename      "sea_test1"       hydrophone         //string
   VAR      debug_stuff         OFF               beamformer         //report
   VAR      interesting_vars    FRAMES,gain,2,5   beamformer(0)      //report
   VAR      FUNNEL.display.elem 4,47,81,89                           //funnel
```

Figure 5. Example of a Database Startup file.

The type of value which can be assigned to database variables are integer, floating-point, string, *report,* and *funnel*. Integers must be specified either as an integer without a decimal point (e.g., 100) or as TRUE or FALSE. Floating-point values contain a decimal point (e.g., 1500.0) or an exponent (e.g., 1e+3) or both; their type is internally represented as a double. Strings are enclosed in double quotes (e.g., “sea_test1”). *report* is a special type which is specified either as ON or OFF or as a list of four components: FRAMES, *portname*, *startframe*, and *endframe*. *funnel* is a special type which is specified as a list of integers. The *report* and *funnel* types will be explained in more detail in later sections. With the exception of the special types, structures or arrays cannot be assigned through the user interface to variables in the database.

The rules for making a Database Startup file are as follows:

1. Tabs and blanks are white spaces and are used to delimit lines into tokens interpreted by the parser. Tokens are either key words, identifiers, integers, or strings. Backslash (\) can be used to continue the end of a line. The rest of the line after a double forward slash (//) is ignored as a comment.
2. Tokens are either key words, identifiers, integers, reals, or strings. Key words which can be used in Database Startup files include `VAR`, `TRUE`, `FALSE`, `ON`, `OFF`, `FRAMES`, and `FUNNEL`, and may be written in uppercase or lowercase letters. Identifiers must be an allowable C language identifier, up to 31 characters, and cannot include any of the key words found in Appendix C. Integers must be specified either as an integer without a decimal point (e.g., 100) or as TRUE or FALSE (defined as 1 and 0, respectively). Floating-point values contain a decimal point (e.g., 1500.0) or an exponent (e.g., 1e+3) or both; their type is internally represented as a double. Strings are enclosed in double quotes.
3. Each line starting with `VAR` declares and initializes a variable to be put in the SPE Database. The first and second fields specify the variable name and value. The value specified must be of the type integer, floating-point, string, *report,* or *funnel*. The last field optionally specifies the program or program and instance the variable is intended for. It can be left empty, indicating that the variable is intended for everyone, or contain a program name, indicating that the variable is intended for all instances of a program, or contain the specific program and instance the variable is intended for. Program names must match those used in the System Definition file. If the variable type is *funnel,* then the last field is not specified (left blank).

3.0 PORT-TO-PORT COMMUNICATION

3.1 PARALLEL CONNECTIONS AND DECOMPOSITION

The SPE allows ports to be specified as *striped* or *replicated*. These port types tell the SPE how it should decompose the data it transfers between ports of programs of multiple instances. When the SPE transfers data to an input port that is replicated, then all instances of the receiving program will be given the same data. When the SPE transfers data from an output port that is replicated, then each instance of the sending program must provide the same data. When the SPE transfers data to an input port that is striped, then each instance will be given a different portion of the data (can be overlapping). When the SPE transfers data from an output port that is striped, then each instance of the sending program will provide a different portion of the data (cannot be overlapping). The SPE can transfer data between ports of similar or dissimilar types.

The size of the data communicated over a striped or replicated port must be defined by the user in two dimensions, rows and columns. The data do not actually have to be two-dimensional, but to the SPE it must be described as such (i.e., [1][1], [1][5], and [5][1] are valid). When the SPE transfers data to a striped port of a program of multiple instances, it divides the data along its row dimension. The data are not decomposed across the column dimension. The algorithm divides the data as equally as possible into row-contiguous portions (see Appendix A.). Figure 6 shows how a 6×4 array of data would be striped across programs of 3 and 2 instances.

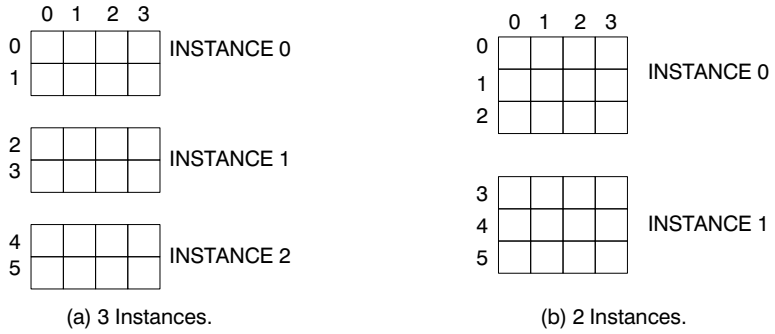


Figure 6. Striped 6x4 array.

When communicating data between programs, the SPE must know how the data are decomposed at the sending and receiving ends of a connection. For a simple two-port connection, the SPE must be able to handle eight basic types of connections: striped–striped, striped–replicated, replicated–striped, replicated–replicated, striped–transposed–striped, striped–transposed–replicated, replicated–transposed–striped, and replicated–transposed-replicated. Figure 7 shows what the communication paths might be for each type of connection. The examples show the sending program running on 3 nodes and the receiving program running on 2 nodes. The data which are communicated are in a 6×4 array. The extent of the data communicated on each path is shown as [*rows*][*columns*].

One can see that even for these simple cases, the level of detail is quite complex. The SPE for each program instance must know where it is sending or getting its data and how it will scatter or gather its data. Each instance within a program will operate differently to communicate its portion of the data. Also, if the sending or receiving programs are scaled to run on a different number of nodes, or if additional ports are added to the net, then the paths will change.

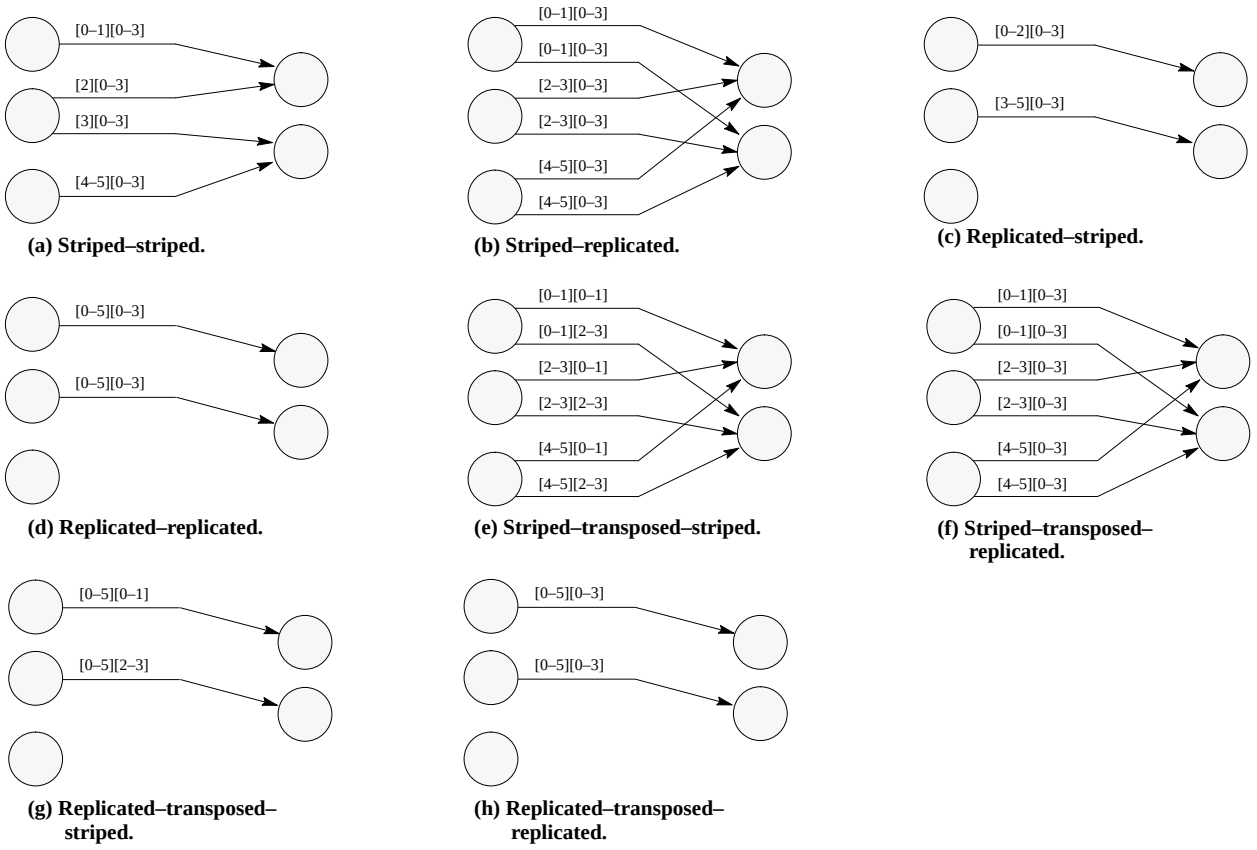


Figure 7. Internal message paths for communicating a 6x4 (t row by 4 column) array.

A program should not have to deal with this level of detail, and indeed this is something which is provided for and kept hidden by the SPE. A program should not be concerned with where it is getting or sending its data, how the data is decomposed at the other end, or even what instance of the program it is. The only thing a program should need to know is what portion of the data it works on and must produce.

Output and input ports can also be connected through *funnels*. A funnel connects an input port to an output port whose row dimension is greater than its own dimension. The user specifies in a Database Startup file which rows of data are actually passed between the sender and receiver. There are four types of connections which can use a funnel: striped–funnelled–striped, striped–funnelled–replicated, replicated–funnelled–striped, replicated–funnelled–replicated. These are connected with the same type paths as the nontransposed-type connections shown earlier. Funnel and transpose connections cannot be used together.

To specify what rows are connected through a funnel, the user must create a database variable of the name **FUNNEL.***program***.***port*, where *program* and *port* are replaced by the program name and input port name to which the funnel is attached. The user assigns to this variable an array of integers containing the row indices of the data which are connected through the funnel. The number of integers specified must be equal to the row dimension of the input port. Shown below is the funnel database variable which was used in our example system:

```
VAR       FUNNEL.display.elem   4,47,81,89
```

3.2 INPUT PORT FIFO BUFFERS

Another feature that the SPE provides is that two programs interconnected can work on different-size data blocks. The SPE requires that the *row* dimension of each input port on a net agree with the output port

to which it connects, but allows the *column* dimension of each to be different. The SPE can allow this by providing a FIFO buffer on each input port which stores the data when the data are received. The FIFO buffer is a two-dimensional buffer which performs the FIFO operation along the columns dimension of the buffer. Since the communicated data is two-dimensional, the FIFO buffer must also be two-dimensional. Figure 8 shows the operation performed by the FIFO buffer.

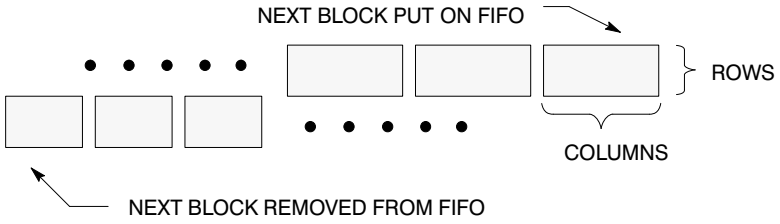


Figure 8. Input port two-dimensional FIFO.

The implementation of two-dimensional FIFO buffers is not straightforward because the physical memory of a computer is accessible in only one dimension (every memory location is accessed by one address). To understand this point, consider figure 9.

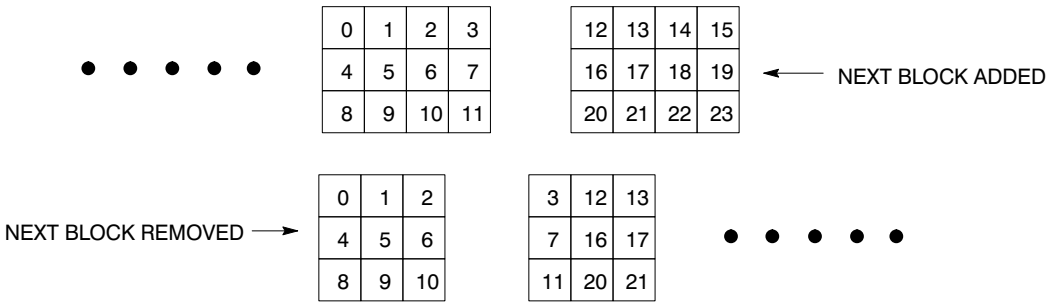


Figure 9. Memory addresses in two-dimensional FIFO.

The 3×4 element boxes at the top of the figure represent the data being put into the FIFO buffer. The 3×3 element boxes at the bottom of the figure represent the data being removed from the FIFO buffer. (Looking back at figure 7a, we see this represents the data sent along the top two paths.) The number in each box represents the address where each element is stored. One can see that if the elements of the input FIFO blocks are stored contiguously in memory, then the elements of the output FIFO blocks will have to be read from noncontiguous memory locations. For instance, the first block removed is read from addresses 0, 1, 2, 4, 5, 6, 8, 9, and 10. Once again, this level of detail is beyond what an application program should be concerned with and is a feature provided by the SPE.

The SPE also allows the blocks of data taken from a FIFO buffer to overlap. The amount of overlap is specified for each input port in the Program Definition files. The *Block_Ovlp* field specifies for a given input port the number of columns each block of data will overlap when removed from the FIFO buffer.

Another important benefit that a FIFO buffer provides is that it allows a program to overlap communication with computation. For example, when a program is working on a block of input data, it can also be receiving in its input FIFO buffer future blocks of data. Thus, when it finishes working on the current block of data, it is ready to start to work on the next. The SPE provides the internal control signals sent between the receiving and sending programs to keep the FIFO buffers full.

3.3 MESSAGE SYNCHRONIZATION

There are still other issues concerning data communication besides how the data are connected or buffered. In parallel processing, where multiple instances of a program work on a problem, the data seen by

each instance of the program must be coherent. Messages received by each instance of a program must be received in the same order. The SPE guarantees that for programs having multiple input ports, the messages received over those ports will be received in the same order by each instance of the program. This is an obvious requirement for a program that receives data from multiple programs which operate asynchronous to each other.

Figure 10 shows the control signals used by the SPE to provide the message-passing synchronization needed between program instances and to buffer the data between sending and receiving programs. The *request* lines from program B to program A indicate that the FIFO buffers in B are ready for more data and also indicate how empty they are. The *sync* lines from B0 to B1 and B2 indicate the order in which B0 has received its messages. B1 and B2 use this information to force their messages to be received by the user's program in the same order.

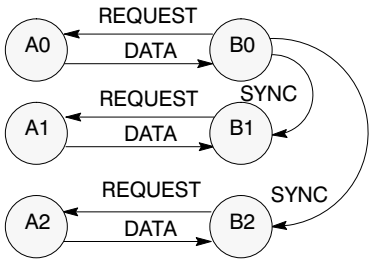


Figure 10. Control signals.

3.4 CONTROL TYPE PORTS

In the Program Definition files, ports can be specified as *control* type. Control ports are provided to allow programs to communicate data which are not a part of the normal data-flow stream of the system. Data sent over a control port do not have specifications for array size, element size, stripe overlap, or block overlap. When the composition of data sent between programs is irregular (cannot be specified as a two-dimensional matrix) or is unknown, then the data must be sent via control ports. Control ports by definition are replicated and can only be connected to other control ports. Control ports are connected like the replicated–replicated connection shown in figure 7d (ignoring the dimensions of the data).

3.5 DATA FLOW

When a sending program puts data on an output port, it blocks until the SPE has sent the data to all the input ports it is connected to. The SPE will send the data to each input port as space is made available in the port's input FIFO buffer. When all the input ports have received the data in their FIFO buffer, the SPE returns control to the sending program.

When a receiving program gets data from an input port, it blocks until the data become available in the port's input FIFO buffer. When enough data have been collected in the FIFO buffer to satisfy the input request, the SPE will transfer the data to the user's buffer and return control to the receiving program.

Internally the SPE controls the flow of data by having the receiving program tell the sending program when it can send more data. Just before the SPE returns control to the receiving program, it makes a decision of whether or not to let the sending program send more data. If the receiving program has room in its FIFO buffer for more data from the sending program, it tells the sending program how much more data to send. While the data are being sent, the SPE returns control to the receiving program, allowing it to work on the current buffer of data. The cycle repeats itself each time the receiving program gets data.

This method of flow control provides the programmer with a simple decentralized method for synchronizing programs. Each program does not have to know about the requirements of the programs it is connected to or have to generate control signals to control the flow of data it consumes or produces. The control signals are handled internally by the SPE. Each program simply receives and produces data as fast as it can go. The double buffer process described above allows a receiving program to work on data while the SPE sends data for the next cycle.

4.0 PROGRAMMING INTERFACE

4.1 MESSAGE INTERFACE

4.1.1 *spe_init(), portsend(), portrecv(), portid(), portinfo()*

The first SPE routine called must be *spe_init()*. This routine blocks until the calling program receives from the SPE loader the port map and a set of database values specific to the program instance. From the port map, the *spe_init()* call determines and allocates the resources needed to perform the message-passing operations used later in the program.

Messages are passed between programs with the *portsend()* and *portrecv()* system calls. These calls perform the special scatter and gather operations needed to transfer data between multiinstanced programs. With them, each instance of a program will send or receive striped or replicated portions of the data (see Section 3.1).

Programs communicate through ports, avoiding the need for a program to know where it is sending or receiving its data. The *portsend()* and *portrecv()* system calls require the caller to provide the *port ID* of the port to send or receive data. The *port ID* of a named port is returned by the *portid()* system call. Port IDs are assigned by the SPE interface and must be used when referring to a port.

The portion of the problem that an instance of a program works on can be found from the *portinfo()* system call. The *portinfo()* routine copies to the supplied address information describing the portion of data which are striped or replicated for the given port and instance. The calling program instance uses this information to determine the portion of data it will work on and to allocate buffers for receiving or sending the data. The calling program must be written so that each instance of it can work on any contiguous-row portion of the data.

These routines and *db_wait()*, which will be described later, represent the minimum set of routines that must be used by a program (both *portsend()* and *portrecv()* do not have to be used). An example program using each of these routines is shown in figure 11.

4.1.2 *Message Interface Example*

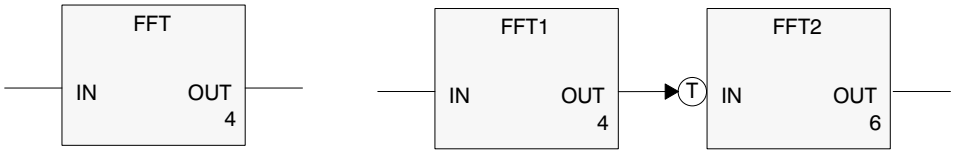
The program illustrated in figure 11 repeatedly performs FFTs on blocks of input data. The input data blocks can be of any size, but must remain fixed over time. Each block of input data is received on port “in” and each FFTed block of output data is sent to port “out”. The program repeats itself forever until the SPE system shuts down.

The program is written so that it can be implemented over any number of instances. Resources, such as the buffer space used to receive input messages, are allocated at run time. Careful use of the *portinfo()* routine is critical to developing a flexible general-purpose program. The program is written so that it can work on any size data block (rows vs. columns), thus maximizing the reuse of the software.

Figure 12 shows how quickly an application can be built by reusing software. The two-dimensional FFT in figure 12b was constructed by simply connecting two copies of the one-dimensional FFT through a transposed connection. No new software was developed.

```
/* File: fft.c
 *
 * Description: Performs FFTs on rows of input matrix (rows x columns).
 *              The FFT size is equal to the number of columns in the matrix.
 *              The rows of the input matrix are striped over the program
 *              instances.  For example if the input matrix is 100 x 128 then
 *              a 128-pt FFT will be performed on each row of the matrix.
 */

#include <spe.h>

long          ii, no_rows, fft_size, port_id, in_pid, out_pid, status;
COMPLEX      *buffer;
size_t        buffer_size;
PORT_INFO in_port_info;

void main()
{
    /* Initialize the SPE interface */
    spe_init();

    /* Register and assign database variables here. */

    db_wait(); /* Explained in "Database Interface" */

    /* Get the port IDs of ports "in" and "out" */
    in_pid = portid("in");
    out_pid = portid("out");

    /* Determine what portion of the problem this instance will work on. */
    portinfo(in_pid, &in_port_info);

    no_rows = in_port_info.end_row -
              in_port_info.start_row + 1;
    fft_size = in_port_info.no_columns;

    /* allocate space for the input data. */
    buffer_size = (size_t)(no_rows * fft_size * sizeof(COMPLEX));
    buffer = malloc(buffer_size);

    /* Loop forever until some other program terminates the run. */
    while (1)
    {
        portrecv(in_pid, buffer, buffer_size, &status);

        for (ii = 0; ii < no_rows; ii++)

            cfft(buffer+ii*fft_size, fft_size, 1); /* buf,size,1=forward */

        portsend(out_pid, buffer, buffer_size);
    }
}
```

Figure 11. Example FFT program illustrating the use of the basic SPE routines.



(a) One-dimensional FFT.

(b) Two-dimensional FFT.

Figure 12. Reuse of an SPE program.

4.1.3 *portwait(), portprobe()*

In the example given in figure 11, the program waits for data on a single port. To wait for data from multiple input ports (when the order of the messages is not known ahead of time), the program must use the *portwait()* and *portprobe()* routines. The *portwait()* routine blocks until a message is ready to be received on one of the input ports. Then when a message is available, the *portwait()* routine returns with the port ID of the pending port. The *portwait()* routine always returns the port IDs of the input messages in the order they were received. All instances of a program are guaranteed to receive the input messages in the same order.

The *portprobe()* routine determines whether a message on a selected input port is ready to be received. The programmer supplies the port ID of the input port to be checked. If port ID is –1, then all input ports are checked. The *portprobe()* routine immediately returns a long value, indicating whether the selected port has a message available to be received. If the programmer has selected a specific port and a message is available on that port, then *portprobe()* returns the port ID of the selected port. If the programmer has selected all input ports to be checked and a message is available on one or more of the input ports, then the port ID of the message which was available first is returned. If a message is not available to be received, the *portprobe()* routine will return a –1.

The example program illustrated in figure 13 shows how one would use the *portwait()* routine. The program does not know ahead of time the order in which messages become available over ports “in1” and “in2.” However, it does know ahead of time that when a message becomes available on port “in2” another will soon follow on port “in3’ (for instance, these messages may be sent from the same program). In figure 13, the *portwait()* routine is used to block the program until a message is available to be received, and then *if–else* statements are used to determine from which port to get the message. When a message becomes available on port “in1,” it is received and processed. When a message becomes available on port “in2,” it is received and processed along with the message from port “in3.” The program does not necessarily have to receive and process messages in the order in which they become available. In the period that the messages on ports “in2” and “in3” are received, a message on port “in1” may have become available.

```
/* Get the port IDs of each input port. */
in1_pid = portid(”in1”);
in2_pid = portid(”in2”);
in3_pid = portid(”in3”);
...
while (1)
{
    /* Wait on a message from any input port. */
    pid = portwait();

    if (pid == in1_pid)
    {
        portrecv(in1_pid, buffer1, buffer1_size, &status);
        ...
    }
    else if ( pid == in2_pid)
    {
        portrecv(in2_pid, buffer2, buffer2_size, &status);
        portrecv(in3_pid, buffer3, buffer3_size, &status);
        ...
    }
    else ...
}
```

Figure 13. Usage of *portwait()*.

4.1.4 *portexits(), portisconnected()*

A program which has been designed for general use may not know ahead of time how many input and output ports it may actually have, or if it does, it may not know whether they are actually connected. The

routines *portexists()* and *portisconnected()* can be used to determine these qualities. The *portexists()* routine returns a boolean value indicating whether the named port exists. The *portisconnected()* routine returns a boolean value indicating whether the named port is connected. Both routines must be supplied with the string name of the port of interest. Data sent to a disconnected port will be dropped. Trying to receive data on a port which is not connected will cause the program to hang.

4.1.5 *porteos()*

A program can send to an output port an end-of-stream (EOS) mark, indicating that the program will temporarily or permanently stop the flow of data to that port. The EOS mark is detected by a receiving program from the *status* argument of the *portrecv()* routine. The EOS mark can be used to determine when a system is finished processing, to reroute the flow of data through a system, or to reconfigure the ports attached to a net.

Figure 14 shows how a typical data-flow system might be connected. Program A reads data from an input file, program B processes the data, and program C writes the processed data to an output file. Each program executes a loop which receives, processes, and produces frames of data. The system will run until program C writes to the output file the last frame of data which program A produces and program B processes. When program C writes the last frame of data to the output file, it will then initiate system termination, causing all the programs to exit.

For this to happen, program C must be able to determine when it has received the last data that it will write. If this information is not embedded in the data, then it must use some out-of-band technique to determine the end of the data. For this reason, the EOS mark is provided to indicate that the end of a stream has been reached. It provides an out-of-band way to tell the user that the last piece of data has been read.

The EOS mark is used as follows: When program A has finished reading the input file and has sent the last frame of data to program B, it sends an EOS mark to program B by calling *porteos()*. Program B detects the EOS mark from the status information returned by the *portrecv()* call and in turn calls *porteos()* to send the EOS mark to program C. Finally, program C detects the EOS mark from the status information returned by the *portrecv()* call, closes the output file, and initiates system shutdown.

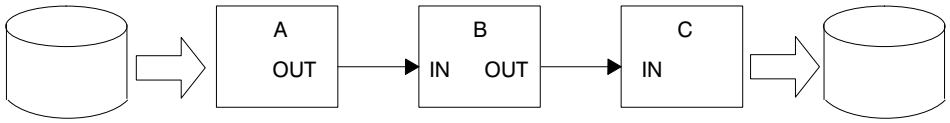


Figure 14. EOS is daisy-chained through programs A, B, and C.

4.1.6 *portbos(), portrecvbos(), portreconfigure()*

A program can send to an output port a beginning-of-stream (BOS) mark, which will be sent to all the input ports to which the net is connected. The BOS mark is used to restart the flow of data through a net or to reconfigure the ports to which the net is connected.

The BOS mark is sent to an output port by calling *portbos()*. *portbos()* must follow *porteos()* and must be called before the flow of data is restarted to a given port. The caller supplies to *portbos()* a port configuration string which will be sent along with the BOS mark to the connected input ports. The port configuration string tells the downstream programs how to reconfigure their input port.

The BOS mark is detected by a receiving program from the status argument of the *portrecv()* routine. The *portrecv()* routine copies the port configuration string, sent along with the BOS mark, to a buffer provided by the caller. The receiving program uses the configuration string to reconfigure its input port.

The *portbos()* routine can be used to restart a data stream in the same way that the *porteos()* routine is used to stop a stream. For example, in figure 14, if program A wants to restart the stream, it would send a BOS mark to program B by calling *portbos()*. Program B detects the BOS mark from the status information returned by the *portrecv()* call and in turn sends the BOS mark to program C. Finally program C detects the BOS mark from the status information returned by the *portrecv()* call.

After a stream to or from a port is restarted (BOS) and before a program can reuse a port, the program must reconfigure the port. A program reconfigures a port with the *portreconfigure()* routine. The program supplies to the *portreconfigure()* routine the string name of a valid configuration for the port as defined in the Program Definition file. When the *portreconfigure()* routine is called, the SPE reallocates buffers and reestablishes connections to the ports to which it is connected. The *portreconfigure()* routine can only be called after a program receives an EOS mark and BOS mark on a port. Valid configurations for ports can be passed along with the BOS mark, providing an easy way for a program to get the information. The *portreconfigure()* routine can also accept a null string (**""**), which tells the SPE to use the previous configuration.

After all programs of ports on a given net have called *portreconfigure()*, the SPE will redo the decomposition for each port and establish new internal communication paths connecting those ports (see Section 3.1). The SPE will make sure that the new port definitions are consistent with each other, using the same requirements as those for the System Definition file (see Section 2.1). Because the SPE redoes the decomposition for each port, each program and instance affected will have to recall the *portinfo()* routine to find out what portion of the new problem it will work on. Each will have to *free()* the memory used by the old message buffers and *malloc()* new memory for the new message buffers.

Figure 15 is an example of an upstream program reconfiguring one of its output ports to handle "big FFTs." The program calls in order the *porteos()*, *portbos()*, *portreconfigure()*, and *portinfo()* routines. The program reconfigures the output port so that it will now handle big FFT-size data blocks. The string message sent in the *portbos()* routine tells the downstream programs how to reconfigure their input port.

```
long       out_pid;
PORT_INFO  out_port_info;
...

/* Send EOS mark to downstream process. */
porteos(out_pid);

/* Send BOS mark to downstream process.  Tell it we're reconfiguring the
 * system to "big FFTs".
 */
portbos(out_pid,      /* port ID */
        "big_ffts");  /* new port configuration */

/* Reconfigure our output port */
portreconfigure(out_pid,     /* port ID */
                "big_ffts"); /* New port configuration. */

/* Determine what portion of the problem this instance will work on. */
portinfo(out_pid, &out_port_info);
```

Figure 15. Reconfiguring an output port.

Figure 16 is an example of a corresponding downstream program reconfiguring one of its input ports to match the upstream port to which it is connected. The program calls in order *portrecv()*, *portrecv()*, *portreconfigure()*, and *portinfo()* to get the EOS mark, to get the BOS mark, to reconfigure the input port, and to determine the new portion of the problem it will work on. After calling the *portinfo()* routine, it frees memory used by the old input buffer and allocates memory for the new input buffer. The program must also recompute (not shown) the number of rows of data it will do next and the new FFT size.

```
long           in_pid, status;
CFGNAME_TYPE   in_port_cfg
PORT_INFO      in_port_info;
...

/* Get next data buffer from upstream process. */
portrecv(in_pid, in_buf_ptr, in_buf_size, &status);
if (status == EOS)
{
    /* Get BOS mark from upstream process. */
    portrecv(in_pid, in_port_cfg, 0, &status);
    if (status != BOS)
    {
        /* ERROR: Tell user that BOS didn't follow EOS. */
    }else
    {
        portreconfigure(in_pid,       /* Port ID */
                        in_port_cfg); /* New port configuration */

        /* Determine what portion of the problem this instance will work on. */
        portinfo(in_pid, &in_port_info);


        /* free memory used with old configuration. */
        free(in_buf_ptr);

        /* allocate memory used in new configuration. */
        in_buf_ptr = malloc(...);
        ...

    }
}
```

Figure 16. Reconfiguring an input port.

4.2 DATABASE INTERFACE

As described earlier, the SPE provides a global database to store symbolic names with their associated values. Variables can be stored and read from the database through either a user or program interface. This section describes the program interface.

The program interface, unlike the user interface, does not consider a variable to have a specific destination. That is, a program cannot specify that a variable should contain different values for different programs or instances of programs. This is consistent with the SPE philosophy that a program need not know about the existence or requirements of other programs in a system.

A program interfaces to the SPE database by first *registering* each variable that it will access from the database. When a program changes a variable's value in the database, then a copy of that variable propagates to all programs which have registered for it. This method for maintaining global data was chosen to maximize system performance. Alternatively, the global database could have been designed so that each time a a variable was stored to the database, it propagated to all the programs, regardless of which program wanted it. Or even worse, it could have been required that each time a program wanted to use a variable in the global database, it would have to make a request. In either case, the global database manager increasingly becomes a bottleneck as the number of programs in a system increases.

When the SPE interface on a program receives a new value for a database variable, it waits before updating the local copy of the variable. It must wait to make sure that when the local copy is changed, the program is not in the middle of accessing it, because certain variables, such as strings or structures, may require atomic access. Also, it must wait to make sure that each instance of a program sees the local copy changed at the same time. For this reason, the following algorithm is used to determine when the local copy is updated: If the program is executing on one node, then the local copy of the variable will be

updated during the next SPE system call. If the program is executing on multiple nodes and has input ports, then the local copy of the variable will be updated inside the next *portrecv()* system call. If the program is executing on multiple nodes and has only output ports, then the local copy of the variable will be updated inside the next *portsend()* system call. *portrecv()* and *portsend()* are allowable update points because the SPE receives database variables as if they were port messages. It guarantees that database variable updates and port messages are received in the same order by each instance of a program.

4.2.1 *db_register(), db_set(), db_wait()*

Three routines are used by a program to interface to the global database. A program calls the *db_register()* routine to tell the database manager that it is interested in a variable. The program then supplies to the routine the string name of the database variable, an address in memory where the local copy of the variable will be maintained, an enumeration indicating the type of variable that it expects, and the size of the variable in bytes. The contents of the local copy of the variable are not sent to the database manager. When a program registers a variable, the global database manager sends the value of the variable to the program if it has already been set. If the program has already called the *db_wait()* routine (described later), then the value is immediately copied to the program's local copy of the variable. If the program has not yet called the *db_wait()* routine, then the variable is updated later when *db_wait()* is called.

A program sets the value of a database variable by calling the *db_set()* routine. It supplies to the routine the string name of the database variable, the address in memory where the value will be copied from, an enumeration indicating the type of variable being stored, and the size of the variable in bytes. When the value of a database variable is set, its value is propagated to all programs which have registered for it.

After a program has registered or set all database variables critical to system startup, it calls the *db_wait()* routine. This routine is a system-synchronizing routine which waits until all programs in the system have also called *db_wait()*, indicating that they too have registered or set database variables critical to system startup. This routine *must* be called regardless of whether a program registers or sets database variables. The *db_wait()* routine also updates local copies of the database variables which were set after they are registered.

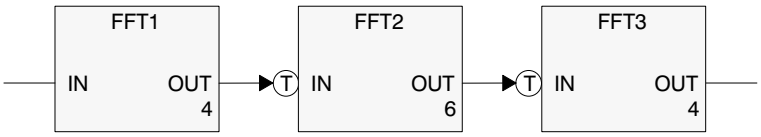
The example in figure 17 shows how one might use the global database to make the FFT program more general purpose. The program uses the global database variable "forward_fft" to determine whether it should perform a forward or reverse FFT. The variable would be set from a Database Startup file or Program Definition file where it could be set differently for each usage of the program. The FFT program (or any program in the system) must make sure that it does not set a value to this database variable, thus propagating the same value to all usages of the program (different programs may be asked to do different-size FFTs). So in this case, it might be wise to make the database variable name more unique to the function of the program.

Figure 18 shows how one might use the new FFT program to build a simplified beamformer. Also shown is the Database Startup file that controls whether each program does forward or reverse FFTs. One can see how quickly a system can be built by reusing software.

Extending this concept of reusable software, one might build a general-purpose processing module which could perform any of the functions found in a standard vector-processing library. A global database variable would determine how each usage of the processing module within a system would function. For instance, the string database variable "libxx_function" could be used to determine if the libxx.c program would perform an FFT, correlation, or vector magnitude function. From the Database Startup file, one could specify a different function for each use of the program.

```
void main()
{
    ...
    BOOLEAN       forward_fft = TRUE; /* Default: forward FFT */
    ...
    spe_init();

    /* "forward_fft" will be set in the Database Startup file. */
    db_register("forward_fft", &forward_fft, DB_INT, sizeof(BOOLEAN));

    /* Wait for other programs to register or set database variables.
     * Update local copies of the database variables. */
    db_wait();
    ...

    while (1)
    {
        portrecv(in_pid, buffer, buffer_size, &status);

        for (ii = 0; ii < no_rows; ii++)
            if (forward_fft)
               cfft(buffer+ii*fft_size, fft_size, 1); /* Forward FFT */
            else
               cfft(buffer+ii*fft_size, fft_size,-1); /* Reverse FFT */

        portsend(out_pid, buffer, buffer_size);
    }
}
```

Figure 17. Using a global database variable.



(a) System diagram.

```
// File: database/beamformer
//
// Key Word Name             Type(Value)                    Program(instance)
// ------------------------------------------------------------------------
   VAR      forward_fft      TRUE                           fft1
   VAR      forward_fft      TRUE                           fft2
   VAR      forward_fft      FALSE                          fft3
```

(b) Database Startup file.

Figure 18. Program reuse controlled by the global database.

4.3 REPORT INTERFACE

Debugging a parallel application requires that the user deal with multiple programs and multiple instances of programs. Using the traditional *printf()* statement to trace the progress of an application is not practical because of its replicated use when called from programs implemented on multiple nodes or from common modules used by multiple programs. Because of its replicated use, when *printf()* writes to standard output, the user gets more information than bargained for (e.g., 50 repetitions of the same message), and in addition, does not know which program or program instance has generated each output. (Also, on the Paragon, when more than one *printf()* is used simultaneously, their results fragment and mix to the standard output.) What is needed instead is a routine which acts like *printf()* but which conditionally executes based on conditions that the user can control.

4.3.1 *report()*

The SPE provides to the programmer the *report()* system call. The *report()* system call functions the same as the *printf()* system call except that it requires one extra argument. The first argument to *report()* specifies a *report category variable* in the global database that *report()* will use at run time to determine if it should actually write the data to standard output. Report category variables are created by the user, through Database Startup files, to control which *report()* calls write to standard output. Each use of *report()* can refer to a different report category variable, but through careful selection of categories and placement of *report()* calls, one can create an effective debugging environment. The other arguments to *report()* look the same as that used in *printf()*.

Report category variables are created by the user. Through them, the user tells the application which categories of reports it wants to see, for which programs and instances, and for what range of time (time is dictated by range of messages over a specific port). Shown below is how a report category variable is internally constructed in the SPE:

```
typedef enum {OFF,ON,FRAMES} MODE;
typedef struct {
MODE        mode;
char        port_name[32];
long        start_frame;
long        end_frame} REPORT;
```

The *mode* field tells *report()* which mode to use to determine if it should write to standard output. If mode is `OFF`, then *report()* will not generate output. If mode is `ON`, then *report()* will generate output for the selected program and instance each time it is called. If mode is `FRAMES`, then *report()* will generate output between the times determined by the *port_name*, *start_frame*, and *end_frame* fields. *port_name* is the name of a port for the target program and *start_frame* and *end_frame* specify the message counts which delimit the time that the report will be generated. Most of the time, the user will indicate that reports are not desired. *report ()* calls which use a report category variable which has not been defined will not generate output.

The user creates report category variables by specifying them in the Database Startup files. Figure 19 is an example of how one might specify a report category variable in a Database Startup file.

```
// Key Word Name          Type(Value)        Program(instance)
-------------------------------------------------------------------
    VAR   interesting_vars  FRAMES,gain,2,5        beamformer(0)
```

Figure 19. Specifying *report()* output using `FRAMES` mode.

This line says that we want to see output which refer to “interesting_vars” from the *report()* routines called from instance 0 of the beamformer program and called between the times that the gain port receives its 2nd and 5th message. The user can set a different value for “interesting_ vars” to each program and instance in the system, or the same value to all instances of a specific program in the system, or the same value to all instances of all programs in the system.

For this report category variable to be effective, the beamformer program would have put *report()* calls after places where it computes these interesting variables. For example, a portion of the beamformer program might look like:

```
speed_of_sound = ...
report("interesting_vars","speed_of_sound=%f",speed_of_sound);
...
no_bad_sensors = ...
report("interesting_vars", no_bad_sensors=%d", no_sensors);
```

As a result, when these calls are executed, meeting the conditions specified in the report category variable, the *report()* will generate output. For example, the following output might appear:

```
REPORT:beamformer(0):gain:frame=2, interesting_vars, clk=87.887,node=24
------------------------------------------------------------------------
speed_of_sound=1588.1
REPORT:beamformer(0):gain:frame=2, interesting_vars, clk=87.889,node=24
------------------------------------------------------------------------
no_bad_sensors=0
```

When the *report()* routine writes data to standard output, it provides a header portion that indicates the name of the report category variable, the name of the calling program, the instance of the calling program, the time at which it occurred, and the physical node number.

The report category variables “error”, “warning”, and “info” are predefined. The user cannot set values for these variables. When a program uses them in a *report()* call, it forces the formatted data to be written to standard output. Also, the SPE generates a summary report at system termination that indicates how many times each of these predefined variables are used. Shown below are example uses of these categories:

```
/* Force report() to generate output. */
report("info","Beginning Initialization");
...
if (speed_of_sound > 2000)
    report("warning","speed of sound out of range");
...
```

Many of the SPE system calls have associated report category variables which can be set by the user. They can be used to determine when system calls are entered and exited, thus tracing the execution of a program. Other report categories can be used to determine when memory is allocated with *malloc()*, when files are open and closed, or to report performance monitoring statistics (see Appendix B).

4.4 PERFORMANCE MONITORING INTERFACE

4.4.1 *monitor_on(), monitor_off()*

The SPE provides two simple routines, *monitor_on()* and *monitor_off()*, which the user can use to monitor the performance of sections of code within an application program. The user would use these routines to help find bottlenecks within the application and thus optimize the slow programs of an application or reallocate the hardware resources to the application programs.

The *monitor_on()* and *monitor_off()* routines are placed around sections argument of code that the programmer wants performance statistics on. The programmer supplies a string arugment to the monitor routines that identifies the section of code to be monitored. The monitor routines keep track of how many times each section of code is entered, how much accumulated time is spent in each section, the minimum and maximum times spent in each section, and the accumulated number of operations performed in each section.

At the end of a run, the user can view the information recorded by each of the monitors. For an example of how to use the monitors and view the results, see the entry for *monitor_on()* in Appendix D.

5.0 USING SPE

5.1 COMPILING AND LINKING AN SPE PROGRAM

When compiling and linking an SPE program on the Intel Paragon, you must use the *–nx* switch. To see the effects of this switch, read the Paragon User's Guide manual. When linking an SPE program, you must link in the library *libspe.a*.

For example, the following command line compiles and links the file *myprogram.c* to create an executable file called *myprogram*:

```
% cc –nx –o myprogram myprogram.c libspe.a
```

5.2 RUNNING AN SPE SYSTEM

A user loads and runs an SPE application by executing `spe`. The usage for `spe` is:

```
%  spe –pn partition –on 0 –s sys_def_filename
       [[-d database_startup_filename]...] [-l log_filename]
       [-ident] [–identall] [–noload] [-portmap]
       [[–Dname]...] [[–Dname=def]...]
```

The arguments *–pn partition –on 0* are arguments to the Paragon *application(1)* command, which say that the `spe` program will run on node 0 of partition *partition.* The remainder of the arguments, *sys_def_filename [–ident][–identall][–noload][-portmap][–l log_filename] [[–d database_startup_filename]...][[–Dname]..] [[–Dname=def]..]*, are passed to the `spe` program after it is loaded on node 0.

The *sys_def_filename* argument is required. This is the name of the System Definition file that `spe` will read to start the application. `spe` then begins to load the programs specified in the System Definition file onto the target nodes of the hardware. From the System Definition file, `spe` reads the Program Definition files and builds and downloads a unique port map to each program instance in the system. `spe` then reads the Database Startup files as specified on the command line and in the Program Definition files and initializes the SPE database. Once this is done, the programs are ready to run.

As an example, to run the system described in figure 3 you would execute:

```
%  mkpart –sz 30 mypart
%  spe –pn mypart –on 0 –s system/receiver -d database/receiver
```

Appendix A

STRIPE ALGORITHM

The algorithm used to compute the range of rows that will be striped over each instance of a program is as follows:

```
if (instance < no_rows % no_instances)
{
    start_row = instance * (no_rows / no_instances + 1);
    end_row   = start_row + (no_rows / no_instances);
}
else
{
    start_row = instance * (no_rows / no_instances) +
                  (no_rows % no_instances)
    end_row   = start_row +  (no_rows / no_instances)– 1
}
```

As an example, if a 100x200 array is striped over 3 instances, then instance 0 will have rows 0 to 33, instance 1 will have rows 34 to 66, and instance 2 will have rows 67 to 99. The columns dimension of the array does not affect the striping.

Appendix B

PREDEFINED REPORTS

spe_init

```
spe_init()(entering):
spe_init()(exiting): freemem = %d
```

spe_idle

```
spe_idle(): Program has gone into idle state.
```

spe_terminate

```
spe_terminate(): Starting termination.
spe_terminate(): Finishing termination.
```

spe_malloc

```
spe_malloc(): ERROR: Could not malloc %d bytes of memory for
      'purpose_str'.  %d bytes of memory left.

spe_malloc(): Malloced %d bytes of memory at address %d for
             'purpose_str'.  %d bytes of memory left.
```

db_set

db_set(): Set the value of *'variable_name'*.

db_register

db_registered(): Registered *'variable_name'*.

portprobe

```
portprobe()(entering): Checking for message on any port.
portprobe()(exiting): Message available on port "portname".
```

portwait

```
portwait()(entering): Waiting for an available message on any port.
portwait()(exiting): Message available on port "portname"
```

portrecv

```
portrecv("portname")(entering): Waiting to receive message.
portrecv("portname")(exiting): Received message.
```

portsend

```
portsend("portname")(entering): Waiting to send message.
portsend("portname")(exiting): Sent message.
```

show_monitors

Appendix C

KEY WORDS

```
BUFFER
CONTROL
EXCLUDE
FALSE
FRAMES
FUNNEL
INPUT
NA
NET
OFF
ON
OUTPUT
PORT
PROGRAM
REPLICATED
STRIPED
TRANSPOSE
TRUE
VAR
```

Appendix D

PROGRAMMING CALLS

db_register()
db_set()
db_wait()
monitor_on(), monitor_off()
port_discard_data() (no man page)
porteos(), portbos()
portexists(), portisconnected()
portid()
portinfo()
portcount() (no man page)
portname()
portprobe()
portreconfigure()
portrecv()
portsend()
portwait()
programinfo()
report()
report_enabled()
spe_clock()
spe_idle()
spe_init()
spe_freemem()
spe_malloc(), spe_free() (no man page)
spe_terminate()
spe_terminate_define()
write_mat_file() (no man page)
write_sparse_mat_file() (no man page)

DB_REGISTER()

db_register(): Tell the database manager that we are using a variable of a given name and size.

Synopsis

```
#include <spe.h>

void db_register(
        const char      *name,
        void             *address,
        DB_TYPE          type,
        size_t           size);

typedef enum DB_TYPE {
        DB_INT,
        DB_FLOAT,
        DB_DOUBLE,
        DB_STRING,
        DB_REPORT,
        DB_FUNNEL,
        DB_USER_DEFINED
} DB_TYPE;
```

Parameters

name	is the symbolic name of the variable to be registered. *name* must be 31 characters or less.
address	is the address in memory where the variable will be maintained.
type	is an enumeration indicating the type of variable expected.
size	is the size in bytes of the variable to be maintained.

Description

Tell the database manager that a variable of the given *name* and *size* will be used. Each program which uses a database variable must register for it. All programs registering for the same variable must give the same value for the variable *type* and *size*. Once the variable is registered, its current value will be maintained in the location provided by *address*. The variable will be *delay-updated* after a program calls the **db_set()** routine. So that the update appears atomic and synchronous, the update will occur in the recipient programs when the next SPE routine which allows updates is called. Since programs are synchronized to the point of input messages, when a program is run on multiple instances, the variable will be updated when the next **portrecv()** routine is called. If the program has no input ports the variable will be updated when the next **portsend()** routine is called. If the program is run from only one instance, then the variable will be updated during the next SPE call.

Errors

The SPE system will terminate and produce an error message if the *type* or *size* arguments disagree with what is stored in the database.

DB_SET()

db_set(): Copy the value at the specified address to the named variable in the database.

Synopsis

```
#include <spe.h>

void db_set(
    const char      *name,
    void            *address,
    DB_TYPE         type,
    size_t          size);

typedef enum DB_TYPE {
    DB_INT,
    DB_FLOAT,
    DB_DOUBLE,
    DB_STRING,
    DB_REPORT,
    DB_FUNNEL,
    DB_USER_DEFINED
} DB_TYPE;
```

Parameters

name is the symbolic name of the database variable to which a new value will be copied. *name* must be 31 characters or less. The variable must have already been registered with **db_register()**.

address is the address in memory where the value is copied from. Typically this would be different from the address used in **db_register()**, to which the data are copied**.** If the values are the same, then it is possible that not all instances of the calling program would see a variable change value at the same time.

type is an enumeration indicating the type of variable being stored to the database.

size is the size in bytes of the variable to be copied. If the size does not agree with the registered variable, then the SPE system will terminate and produce an error message.

Description

Copy the value at the specified address to the named variable in the database. The database manager *delay–updates* the value to each program which has registered for it (including itself). See **db_register()** for description.

Errors

The SPE system will terminate and produce an error message if the *type* or *size* arguments disagree with what is stored in the database.

DB_WAIT()

db_wait(): Wait until all programs in the system have registered.

Synopsis

#include <spe.h>

void **db_wait**(void);

Description

Wait until all programs in the system have registered, and set all variables critical to startup. This routine is used as a form of synchronization to the database to make sure that all programs have registered variables critical to startup before proceeding. Programs are still able to register variables after **db_wait()**.

Errors

Must be called only once and after **spe_init()** or else the SPE system will terminate and produce an error message.

MONITOR_ON(), MONITOR_OFF()

monitor_on(), monitor_off(): Keep performance statistics on section of code surrounded by these calls.

Synopsis

```
#include <spe.h>

void monitor_on(
        const char      *section_name);

void monitor_off(
        const char      *section_name,
        long             no_ops);
```

Parameters

section_name is the name of the section being monitored. Must be the same in both routines for the section being monitored. *section_name* must be 31 characters or less.

no_ops is the number of operations executed in the section of code being monitored.

Description

These routines are placed around sections of code for which the programmer wants performance statistics. **monitor_on()** and **monitor_off()** are placed, respectively, at the beginning and end of a section of code. The same *section_name* string must be supplied to both. When the **monitor_on()** routine is called, the time on the hardware clock is recorded for the section of code which will be monitored. When the corresponding **monitor_off()** routine is called (has the same *section_name*), the hardware clock is read, and the elapsed time since the **monitor_on()** routine was called is computed. The elapsed time is added to a variable keeping track of accumulated time, and compared to other variables keeping track of minimum and maximum values. Also recorded are the number of times each section of code is entered and the number of accumulated operations performed by each. At the end of a run, the user can view the data recorded by the monitor routines by turning on the ***performance*** report category variable for the interested programs and instances. For instance

```
VAR  performance  ON   beamformer(0)
```

Sections of code surrounded by the **monitor_on()** and **monitor_off()** routines can embed other sections of code being monitored. Also, different sections of code can use the same *section_name*, thus grouping the statistics for those sections. The *no_ops* argument can be set to zero if the user does not care about the ops/sec statistic.

Errors

The SPE system will terminate and produce an error message if **monitor_on()** and **monitor_off()** are not called in order for a given section of code.

Example

```
...
monitor_on(”both”);

/* 128-pt Forward FFT */
monitor_on(”fft”);
cfft(buf, 128, 1);
monitor_off(”fft”,4480); /* 5n*logn = 4480 */

/* 128-pt Inverse FFT */
monitor_on(”ifft”);
cfft(buf, 128, –1);
monitor_off(”ifft”,4480);

monitor_off(”both”,0);
...
```

PORTEOS(), PORTBOS()

porteos(): Sends an end-of-stream (EOS) mark to an output port.
portbos(): Sends a beginning-of-stream (BOS) mark and a string containing a port configuration label to an output port.

Synopsis

```
#include <spe.h>

void porteos(
        long    port_id);

void portbos(
        long    port_id;
        char    *port_cfg);
```

Parameters

port_id is the port ID of the output port to which the EOS mark or BOS mark will be sent. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe()**, and **portwait()** system calls.

port_cfg is a string containing a port configuration label which is sent to an output port. *port_cfg* must be 31 characters or less.

Description

porteos() sends an EOS mark to an output port. The EOS mark indicates that the program will temporarily or permanently stop the flow of data to that port. The EOS mark can be detected by a receiving program from the status argument of the **portrecv()** routine. The EOS mark can be used to determine when a system is finished processing, to reroute the flow of data through a system, or to reconfigure the ports attached to a net. See Section 4.1.5 on how to use **porteos()**.

portbos() sends a BOS mark and a string containing a port configuration label to an output port. The BOS mark and port configuration can be detected by a receiving program from the status argument of the **portrecv()** call and the contents of the receive buffer. The BOS mark is used to restart the flow of data through a net or to reconfigure the ports to which it is connected. See Section 4.1.6 on how to use **portbos()**.

Errors

The SPE system will terminate and produce an error message if **porteos()** and **portbos()** are called out of order or if the *port_id* argument is not a valid output port ID.

PORTEXISTS(), PORTISCONNECTED()

portexists(): Returns a boolean value indicating whether the port exists.
portisconnected(): Returns a boolean value indicating whether port is connected.

Synopsis

```
#include <spe.h>

BOOLEAN portexists(
        char    *portname);

BOOLEAN portisconnected(
        char    *portname);
```

Parameters

portname is the name of the port to check for existence or connectivity. *portname* must be 31 characters or less.

Description

portexists() returns a boolean value indicating whether the named port exists. This routine can be used by a program designed to work with any number of input or output ports. For example, a multiplexing program may not know ahead of time how many input ports it will have to multiplex data from.

portisconnected() returns a boolean value indicating whether the named port is connected to a net. This routine can be used by a program designed to allow partial connectivity to its ports. It will allow the program to avoid reading or writing to ports not connected to a net.

PORTID()

portid(): Returns the port ID for the named port.

Synopsis

```
#include <spe.h>

long portid(
        char    *portname);
```

Parameters

portname must be the name of one of the ports specified in the Program Definition file of the calling program. If the named port does not exist, then the SPE will terminate the run and produce an error message. *portname* must be 31 characters or less.

Return Value

Returns the port ID for the named port.

Description

Returns the port ID for the named port. The SPE system calls use port IDs to receive or send data over the specified ports.

Errors

The SPE system will terminate if the named port does not exist in the Program Definition file of the calling program.

PORTINFO()

portinfo(): Copies the configuration information of a port to the address supplied by the caller.

Synopsis

```
#include <spe.h>

void portinfo(
        long            port_id,
        PORT_INFO *port_info);

typedef struct PORT_INFO {
        /* Contains values which are common to each instance. */
        char            name[32];
        char            cfg[32];
        PORT_TYPE type;
        BOOLEAN         is_input;
        BOOLEAN         is_transposed;
        BOOLEAN         is_funneled;
        long            no_buffers;
        long            no_rows;
        long            no_columns;
        long            elem_size;
        long            striped_ovlp;
        long            block_ovlp;

        /* Contains values which are unique to each instance. */
        long            start_row;
        long            end_row;
        long            start_ovlp_row;
        long            end_ovlp_row;
} PORT_INFO;

typedef enum PORT_TYPE {REPLICATED, STRIPED, CONTROL}
PORT_TYPE;
```

Parameters

port_id is the port ID of the port for which information is sought. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe(),** and **portwait()** system calls.

port_info is the address to which the port's configuration information will be copied.

Description

Copies the configuration information of a port to the address supplied by the caller. Portions of the configuration information will be unique to the instance of the calling program. The calling program uses this information to determine which portion of the problem it works on. If *type* is **STRIPED,** then *start_row, end_row, start_ovlp_row,* and *end_ovlp_row* contain valid data which are unique to each instance. If *type* is **REPLICATED** or **CONTROL**, then they are not used. The other fields of the structure always contain valid data and are the same for each instance of a given port.

Errors

The SPE system will terminate and produce an error message if the *port_id* argument is not a valid port ID.

PORTNAME()

portname(): Returns a pointer to the string name of the port for the given port ID.

Synopsis

```
#include <spe.h>
char *portname(
        long    port_id);
```

Parameters

port_id	must be a valid port ID. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe(),** and **portwait()** system calls.

Description

Returns a pointer to the string name of the port for the given port ID.

Errors

The SPE system will terminate and produce an error message if *port_id* is not valid.

PORTPROBE()

portprobe(): Determines whether a message on a selected input port is ready to be received (non-blocking).

Synopsis

```
#include <spe.h>

long portprobe(
        long    port_id);
```

Parameters

port_id is the input port to be checked. Setting this value to –1 checks all input ports.

Return Value

If a message is ready to be received, **portprobe()** returns the port ID of the selected port. Otherwise, it returns a minus one (–1).

Description

Determines if a message on a selected input port is ready to be received. The programmer supplies in the argument *port_id* the ID of the input port to be checked. If *port_id* is –1, then all input ports are checked. **portprobe()** immediately returns a long value, indicating whether the selected port has a message available to be received. If the programmer has selected a specific port (*port_id* is not –1) and a message is available on that port, then **portprobe()** returns the port ID of the selected port. If the programmer has selected that all input ports be checked (*port_id* is –1) and a message is available on one or more of the input ports, then the port ID of the message which was available first is returned. If a message is not available to be received, the **portprobe()** routine will return a –1.

Errors

The SPE system will terminate and produce an error message if *port_id* is not a valid input port.

Report

The database variable ***portprobe*** can be set so that the program instance will write a debug message to standard output when **portprobe()** is called.

When it is called, it will write one of the following statements to standard output. *portname* is replaced with the port the message is available on.

```
portprobe(): Message not available on any input port.

portprobe(): Message not available on port "portname".

portprobe(): Message available on port "portname".
```

PORTRECONFIGURE()

portreconfigure(): Reconfigures the port to one of the alternate configurations found in the Program Definition file.

Synopsis

```
#include <spe.h>

void portreconfigure(
        long    port_id;
        char    *port_cfg);
```

Parameters

port_id is the port ID of the port to be reconfigured. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe()**, and **portwait()** system calls.

port_cfg is a string containing a port configuration label to which the port will be reconfigured. *port_cfg* must be 31 characters or less.

Description

Reconfigures the port to one of the alternative configurations found in the Program Definition file. The port is reconfigured to a port configuration as specified by the argument *port_cfg*. **portreconfigure()** must be called after an EOS and BOS mark are sent to an output port or after an EOS and BOS mark are detected on an input port. See Sec 4.1.6 on how to use **portreconfigure()**.

Errors

The SPE system will terminate and produce an error message if **porteos()**, **portbos()**, and **portreconfigure()** are called out of order when an output port is reconfigured, or if **portreconfigure()** is not called after a receiving program detects an EOS and BOS mark on an input port, or if the *port_id* argument is not a valid port ID.

PORTRECV()

portrecv(): Posts a receive for a message on an input port and blocks the calling process until the receive completes.

Synopsis

```
#include <spe.h>

void portrecv(
        long    port_id,
        void    *buf,
        size_t  len;
        long    *status);
```

Parameters

port_id	is the port ID of the input port on which the message will be received. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe(),** and **portwait()** system calls.
buf	points to the buffer where the message will be received.
len	is the size of the receiving buffer in bytes. This is used as a consistency check by the SPE. The SPE knows from the port map what the message size should be. If the values do not agree, then SPE will terminate the run and produce an error message.
status	indicates whether an EOS or BOS mark has been detected.

Description

Posts a receive for a message on an input port and blocks the calling process until the receive completes. This routine performs a special type of message transfer in which the data received have been combined and collected from multiple instances according to the port map specified for the receiving instance.

This routine can also detect whether the sender has sent an EOS or BOS mark. If an EOS mark is detected, then *status* will contain the predefined *long* value EOS and no data will be copied to *buf*. If a BOS mark is detected, then *status* will contain the predefined *long* value BOS and a string message from the sender will be copied to *buf*. If neither mark is received, then *status* will contain the value zero and a normal message transfer occurs.

Errors

The SPE system will terminate and produce an error message if the *len* argument is not the right size or if the *port_id* argument is not a valid input port ID.

Report

The database variable ***portrecv*** can be set so that the program instance will write a debug message to standard output when **portrecv()** is called and returned.

When it is called, it will write the following statement to standard output. *portname* will be filled in with the port on which the call is waiting.

```
portrecv()(entering): Waiting to receive a message on
                          port "portname".
```

When it is returning, it will write the following statement to standard output:

```
portrecv()(exiting): Received a message on port "portname".
```

PORTSEND()

portsend(): Sends a message to an output port and blocks until the send completes.

Synopsis

```
#include <spe.h>

void portsend(
        long    port_id,
        void    *buf,
        size_t  len);
```

Parameters

port_id	is the port ID of the output port to which the message will be sent. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe()**, and **portwait()**, system calls.
buf	points to the buffer containing the message to send.
len	is the size of the sending buffer in bytes. This is used as a consistency check by the SPE. The SPE knows from the port map what the message size should be. If the values do not agree, the SPE will terminate the run and produce an error message.

Description

Sends a message to an output port and blocks until the send completes. This routine performs a special type of message transfer in which the data are decomposed and sent to specific instances according the port map for the sending instance. When the routine returns, the buffer can be reused.

Errors

The SPE system will terminate and produce an error message if the *len* argument is not the right size or if the *port_id* argument is not a valid output port ID.

Report

The database variable ***portsend*** can be set so that the program instance will write a debug message to standard output when **portsend()** is called and returned.

When it is called, it will write the following statement to standard output. *portname* will be filled in with the port to which the message is sent.

```
portsend()(entering): Waiting to send message to port
                      "portname".
```

When it is returning, it will write the following statement to standard output:

```
portsend()(exiting): Sent message to port "portname".
```

PORTWAIT()

portwait(): Waits until a message is ready to be received and returns the port ID for the message.

Synopsis

#include <spe.h>

long **portwait**(void);

Return Value

Returns the port ID of a message ready to be received. Port IDs are assigned by the SPE system and are returned by the **portid()**, **portprobe()**, and **portwait()** system calls.

Description

The **portwait()** routine blocks until a message is ready to be received on one of the input ports. Then when a message is available, the **portwait()** routine returns with the port ID of the pending port. The **portwait()** routine always returns the port IDs of the input messages in the order they were received. All instances of a program are guaranteed to receive the input messages in the same order. See also **portprobe()**.

Errors

The SPE system will terminate and produce an error message if there are no input ports specified in the System Definition file for the calling program.

Report

The database variable ***portwait*** can be set so that the program instance will write a debug message to standard output when **portwait()** is called and returned.

When it is called, it will the write the following to standard output:

```
portwait()(entering): Waiting for an available message on
                      any port.
```

When it is returning, it will write the following to standard output. *portname* will be filled in with the port name on which the message is available.

```
portwait()(exiting): Message available on port "portname"
```

PROGRAMINFO()

program(): Copies the program information to the address supplied by the caller.

Synopsis

```
#include <spe.h>

void program(
        PROGRAM_INFO   *port_info);

typedef struct PROGRAM_INFO {
        char            name[32];
        long            no_ports;
        long            no_instances;
        long            my_instance;
} PROGRAM_INFO;
```

Parameters

port_info is the address to which the program information will be copied.

Description

Copies program information to the address supplied by the caller. The calling program can use this to determine its symbolic name, how many ports it has, how many instances of the program there are, and which instance it is.

REPORT()

report(): Conditionally writes to standard output formatted data.

Synopsis

```
#include <spe.h>

void report(
        const char      *report_ctg;
        const char      *format;
        ...);

(internal typedef)
typedef struct {
        MODE            mode;
        char            port_name[32];
        long            start_frame;
        long            end_frame
} REPORT;

typedef enum {OFF,ON,FRAMES} MODE;
```

Parameters

report_ctg is the name of the global database variable that determines whether this routine will write to standard output. The database variable is internally typed as **REPORT**. *report_ctg* must be 31 characters or less.

format is the format string which controls how the data are written. It is identical to the **printf()** format string.

Description

This routine conditionally writes formatted data to standard output. It functions identically to **printf()** except that it has an additional argument, *report_ctg*, which it uses to determine whether it will write to standard output. *report_ctg* is the name of a report category variable in the global database. The user creates and manipulates report category variables for use by the **report()** routine. The **report()** routine will write to standard output based on the contents of the report category variable. The following formula is used:

```
if ((mode == ON) ||
    ((mode == FRAMES) &&
     (current_frame(port_name) >= start_frame)  &&
     (current_frame(port_name) <= end_frame)))
```

The user creates report category variables by specifying them in the Database Startup files. From this file, the user can control the contents of report category variables thus affecting which **report()** calls will output data. For a complete description, see Section 4.3.

When the **report()** routine writes data to standard output, it provides a header portion indicating the name of the report variable, the name of the calling program, the instance of the calling program, the physical node number, and the time at which it occurred.

Example

To turn on a report which prints the "interesting" variables computed by instance 0 of the beamformer program, between the times that the "gain" port receives its 2nd and 5th message, the following line should be included in one of the Database Startup files:

```
VAR interesting_vars   FRAMES,gain,2,5  beamformer(0)
```

The beamformer program would have embedded **report()** calls in the program where the "interesting" variables are computed.

```
speed_of_sound = ...
report("interesting_vars","speed_of_sound=%f",speed_of_sound);
...

no_bad_sensors = ...
report("interesting_vars", "no_bad_sensors=%d", no_sensors);
```

The standard output might look like:

```
REPORT:beamformer(0):gain:frame=2,interesting_vars,
clk=87.887,node=24
------------------------------------------------------------------
speed_of_sound=1588.1

REPORT:beamformer(0):gain:frame=2,interesting_vars,
clk=87.889,node=24
------------------------------------------------------------------
  no_bad_sensors=0
```

Predefined Report Variables

The variables **error**, **warning**, and **info** are predefined report variables. When a program uses them in a **report()** call, it forces the formatted data to be written to standard output. The SPE keeps track of how many **report()** calls of each type are made. Example:

```
report("info","Beginning Initialization");
...
if (speed_of_sound > 2000)
    report("warning","speed of sound out of range");
    /* Force this to be printed */
...
```

REPORT_ENABLED()

report_enabled(): Returns a boolean value indicating whether the report category variable is set so that it would cause a **report()** call, using it to generate output.

Synopsis

```
#include <spe.h>

BOOLEAN report_enabled(
        const char      *report_ctg);
```

Parameters

report_ctg is the name of the global database variable that determines whether a **report()** routine would generate output. *report_ctg* must be 31 characters or less.

Description

This routine returns a boolean value indicating whether the *report_ctg* variable is set so that it would cause a **report()** call, using it to generate output. *report_ctg* is the name of a report category variable in the global database. The routine is the same routine that the **report()** routine uses internally. It is provided to the user so that the user can create its own **report()** style routines. See **report()** for more information about its usage.

SPE_CLOCK()

spe_clock(): Returns the elapsed time in seconds since the application started running.

Synopsis

```
#include <spe.h>

double spe_clock(void);
```

Return Value

Returns a double precision value for the elapsed time in seconds since the application started running.

Description

The **spe_clock()** routine measures the time interval in seconds (with a precision of 100 ns) since the application started running. When the SPE starts an application, it sends to each program instance an offset time which the SPE uses (adds the value to **dclock()**) to get the elapsed time since the application started running.

SPE_IDLE()

spe_idle(): Goes to sleep until the system terminates (never returns).

Synopsis

#include <spe.h>

void **spe_idle**(void);

Description

Goes to sleep until the system terminates (never returns). Allows a program to stop running without breaking the connections it has to other programs. Programs which send their data to a program which has gone idle will continue to run (will not hang even though the receiving program is not there to get the data). After the system terminates, an idle program will still be able to report results collected from its performance monitors.

SPE_INIT()

spe_init(): Initializes the SPE interface. Blocks until all programs in an SPE application have called this routine.

Synopsis

```
#include <spe.h>

void spe_init(void);
```

Description

Initializes the SPE interface. Must be the first SPE routine called. This routine blocks until all programs have initialized their SPE interface.

Errors

The SPE system will terminate if a program cannot initialize properly.

SPE_MALLOC(), SPE_FREE()

spe_malloc(): Gets memory just like **malloc()**, but also generates **report()** messages indicating usage.

Synopsis

#include <spe.h>

void ***spe_malloc**(
    size_t *size*,
    char **purpose_str*);

Parameters

size	is the amount of memory in bytes to allocate.
purpose_str	is the string containing the purpose of the allocation.

Return Value

Returns pointer to space allocated.

Description

Gets memory just like **malloc()** but also generates **report()** messages when called and at the end of a run that indicate memory usage or error conditions. If an error condition occurs when **spe_malloc()** is called (i.e., not enough memory), a **report()** is generated and the **spe_terminate()** routine is called. If no error occurs when **spe_malloc()** is called, then a **report()** message is generated (which the user can turn on or off) that indicates the address of the memory allocated, the amount of memory allocated and the amount of free memory left on the node. Also the user can request that at the end of a run, a **report()** message be generated indicating what memory was allocated during the run. The argument *purpose_str* is a string supplied by the caller indicating the purpose of the **malloc**. It is used in each of the **report()** messages generated.

Errors

The SPE system will terminate and produce a report message if there is not enough memory to allocate. The report message will look like:

```
spe_malloc(): ERROR: Couldn't malloc %d bytes of memory for
              'purpose_str'.  %d bytes of memory left.
```

Report

The report category variable ***spe_malloc*** can be set so that the program instance will generate a **report()** each time **spe_malloc()** is called When it is called, it will write to standard output:

```
spe_malloc(): Malloced %d bytes of memory at address %d for
              'purpose_str'.  %d bytes of memory left.
```

The report category variable ***malloc_summary*** can be set so that the program instance will generate a report at the end of the run indicating what memory was allocated during the run. The report will look as follows:

```
purpose_str    address  amount  memory_left   time
-----------------------------------------------
'purpose_str'   %d       %d      %d            %f
...
```

SPE_TERMINATE()

spe_terminate(): Tells the SPE to terminate the application.

Synopsis

#include <spe.h>

void **spe_terminate**(void);

Description

Tells the SPE to terminate the application. The SPE will cause each program in the system to terminate when the next SPE routine is called, or if already in an SPE routine, to terminate immediately. (It does not interrupt what the user's program is currently doing.) Each program will execute an optionally defined user termination routine (see **spe_terminate_define()**), will generate any **report()** summaries which have been requested, and will then exit. Any program in the system can initiate a system termination by calling this routine.

SPE_TERMINATE_DEFINE()

spe_terminate_define(): Specifies a function to be executed when the program terminates.

Synopsis

```
#include <spe.h>

void spe_terminate_define(
        void    (*term_function) (void));
```

Parameters

term_function is the name of function to execute when the program terminates. The function must have no arguments and return no value.

Description

Specifies a function to be executed when the program terminates. This allows a program to execute critical cleanup code (such as closing files) when the program is terminated by some other program.